Applicative Programming

Basic advantage of expressions: can describe complex, tree-like computations without having to map them to explicit linear sequences of instructions and temporaries.

Our initial expression language had only immutable (unchanging) variables, had no I/O facilities, and computed a single result value.

\[
\text{exp} := \text{num} \mid \text{id} \\
\mid \text{`let' id '=' exp 'in' exp} \\
\mid \text{`letfun' id '(' id ')' '=' exp 'in' exp} \\
\mid \text{`if' exp ('<'|'=') exp 'then' exp 'else'} \\
\mid \text{exp ('+'|'-'|'*'|'/') exp}
\]

- Variables are just abbreviations for complex expressions, as in mathematics.

- Expressions can be evaluated with the aid of an environment mapping variables to values. Once set, an environment entry is never changed.

- Evaluation doesn’t require any notion of state.

- Order of evaluation doesn’t matter, except for data dependencies.

Style of programming called functional or applicative.

Suitable when we can conceive of a program as transformer from inputs to outputs. (More later.)
**Imperative Languages**

Most commonly-used programming languages are **imperative**: they consist of a sequence of operations that alter the **state** of the program.

- The **state** includes the values of variables, which can change by means of **assignment** operations.

- The **state** also includes the input/output history of the program, e.g., the contents of files (or virtual files) read or written by the program’s I/O operations.

- Order of evaluation often matters!

Many languages put have a separate syntactic category of **statements** (or **commands**) that includes stateful operations which don’t produce a result value. But expressions can also affect the state (in which case they are said to have **side-effects**) in addition to returning a result.
Stateful Programming

Stateful programming is a good match to underlying Von Neumann machine programs, which are sequences of instructions that modify the contents of registers and memory locations.

- User-program variables are mapped to machine locations.
- User-program operators include primitive machine instructions.

Imperative languages are also suitable for writing reactive programs that interact with the state of the “real world.” Examples:

- Reading mouse clicks and modifying the contents of a display.
- Controlling a set of relays in an external device.
Assignment

The basic primitive stateful operation is typically assignment, which alters a value stored in a location.

Depending on language, assignments are statements (with no result value), or expressions (maybe with result value).

In the simplest form, the location is associated with a simple variable, e.g.,

\[ a := a + 2 \]

(Will use := for assignment, = for equality relational operator. C/C++/Java use =, == respectively: a bad idea, because both form expressions.)

In most languages, the variable name a means different things on the left-hand and right-hand sides.

On the LHS, a denotes the location of the variable a, into which the value of the RHS expression is to be stored.

On the RHS, a denotes the value currently contained in a, i.e., it indicates an implicit dereference operation.

This makes sense when variables are thought of as locations containing values.

But note that the value of a variable may be a pointer (or reference) to some other location, e.g., in C:
L-values and R-values

Many languages allow more general expressions on the LHS of assignments, as long as they denote locations, e.g. array cells or record fields:

\[
\begin{align*}
a.x & := b.y + 2
\end{align*}
\]

The **l-value** of an expression is the location it denotes (if any) when it appears on a LHS; the **r-value** of an expression is its (ordinary) value when it appears on a RHS.

When it is desirable to use the l-value of a variable somewhere other than the LHS of an assignment, special syntax is needed to indicate that the l-value is wanted. C example:

```c
int y = 0;
setto10(int *x) {
  *x = 10;
}
setto10 (&y);
```
ML References

In ML, ordinary variables are immutable, so they do not have l-values. Updatable variables, called references, must be explicitly created as such, and always serve as l-values. The contents of the variable must be explicitly dereferenced:

\[
\text{let val } x = \text{ref } 2 \\
\text{in } x := ![x] + 2 \\
\text{end}
\]

\[
\text{let val } y = \text{ref } 0 \\
\quad \text{fun } \text{setto10 (x: int ref) = } x := 10 \\
\quad \text{in } \text{setto10 y} \\
\quad \text{end}
\]

This is somewhat more verbose, but removes any confusion between l-value and r-value.
Initialization Values

Most languages require variables (and other sources of l-values) to be declared before they are used: gives them a type and scope, and optionally, an initializing expression.

In fact, it is surely a bug to use any variable as an r-value unless it has previously assigned a value. But many languages permit this, resulting in runtime errors.

The simplest fix is to require an initial value to be given for every declared variable. ML requires this for ordinary (immutable) variables, and also for mutable ref variables.

Java takes a slightly more sophisticated approach:

• variables do not need to be initialized at the point of declaration; but

• they must be initialized before they are actually used.

But in any reasonably powerful language, checking initialization before use is an uncomputable problem.
Definite Assignment

So the Java language reference manual carefully details a **conservative**, computable, set of conditions, which every program must meet, that guarantee there will be no uses before definition.

This is called the **definite assignment** property; just defining it takes 16 pages of the reference manual.

Some programs that **do** in fact initialize before use will be rejected because they violate the conditions.

Legal example:

```java
int a;
if (b)
    a = 3;
else
    a = 4;
a = a + 1;
```

Illegal example:

```java
int a;
if (b)
    a = 3;
if (!b)
    a = 4;
a = a + 1;
```
Order of Evaluation

Order of stateful operations affects program semantics.

**Statements** are always explicitly ordered, making these differences obvious.

**Expressions** can also have side-effects, but order of evaluation is often **under-specified** (precedence and associativity don’t always fix order).

ANSI C example:

```c
a = 0;
b = (a = a + 1) - (a = a + 2);
```

Result (1-3 = -2 or 3-2 = 1 ?) depends on compiler whim.

Side-effects are not always obvious:

```c
a = 0;
h (x, y) = x;
f (z) = a := z;
h(a, f(2));  // = 0 or 2 ??
```

Keeping expression evaluation order or argument evaluation order undefined sometimes lets compiler generate more efficient code.

But modern languages (e.g., Java, ML) have moved towards precise definition of evaluation order within expressions (e.g., left-to-right).
Eagerness

So far, we have assumed that function arguments are always evaluated to values **before** the body of the function is entered. (Without side-effects, can’t tell the difference!)

This is called **applicative order** or **eager** evaluation.

- It is standard practice because it is simple to understand and efficient to implement.
- But it can limit expressiveness of function mechanism.

Example: Consider this Java function:

```java
boolean ifnot(boolean c, int x, int y) {
    if (c)
        return y;
    else
        return x;
}
```

Perfectly legal, but may not do what we’d hope, e.g.,

```java
ifnot(s == 0, t / s, 0);
```

will fail trying to divide by zero if s = 0.
Normal Order Evaluation

In languages using “normal order” or call-by-name evaluation, arguments are not evaluated unless and until they are used within the function body. Maybe useful, but:

- Hard to reason about order of side-effects. Hence, ordinarily used only with applicative languages, such as Haskell.

- Inefficient to implement, because caller’s environment must somehow be transmitted to callee.

pseudo-Haskell example:

```haskell
let t = 10
in letfun ifnot (c,x,y) = if c then y else x
    in letfun f(s) =
        let t = 100
            in ifnot (s==0, t/s, t)
            in f (0)
```

works fine and returns 100 rather than 10.

C example: macros!

```c
#define ifnot(c,x,y) ((c)?(y):(x))
ifnot (s==0,t/s,0);
```

(But they don’t offer lexical scoping for non-parameter variables, and can’t be recursive.)